Cloudproxy Nuts and Bolts

John Manferdelli[1]

Overview

Cloudproxy is a software system that provides *remotely authenticated* isolation, confidentiality and integrity of code and data for Hosted Systems preventing attacks from co-tenants and (under modest assumptions) insiders in a remote data center on supporting hardware. To achieve this, Cloudproxy uses two components: a "Host System" (raw hardware, Virtual Machine Manager, Operating System) which provides capabilities described below to a "Hosted System" (VM, Application, Container).

Cloudproxy provides a mechanism, at each level of the software stack, to isolate Hosted Systems, measure and remotely verify the exact software and configuration information constituting the Hosted System and provide security services like sealing that ensures that information (like keys) can be securely provisioned and retrieved only by the correct Hosted System, while isolated, on a supported platform.

A key concept for Cloudproxy is Code Identity and Measurement that is coupled with isolation and secret provisioning. A Host System measures a Hosted System incorporating the actual binary code and configuration information affecting execution resulting in an unforgeable, compact global identity for that code and execution context. Since the Hosted System knows the "identity" of each Hosted System (i.e.- the unforgeable global identity), it can store secrets that only the Hosted System will receive[2]. The Host Systems can also "attest" to statements made by Hosted Systems by incorporating the unforgeable global identity in statements it signs (again with keys only an isolated Host System has access to). The upshot of this is that a Cloudproxy Hosted System can be isolated, maintain secrets only it knows to encrypt and integrity protect all data it receives or sends, and it can securely authenticate itself over an otherwise unprotected network connection and thus employ authenticated public keys tied to its identity that can be relied upon by communicating parties.

Readers can consult [1] for a fuller description. Source code is in [2].

Cloudproxy Principal Names

Principals in Cloudproxy are general and can represent key based principals, machine based principals, and, most importantly, program based principals. Principal names in Cloudproxy are hierarchical, fully descriptive and securely name the principal. For example, a principal rooted

---

[1] John is manferdelli@google.com. Based on work with Tom Roeder (tmroeder@google.com) and Kevin Walsh(kevin.walsh@holycross.edu).

[2] To do this, the Host System must be isolated and have access to secrets only it knows. The foundation for this consists of hardware primitives provides to the "base" Cloudproxy systems it boots.

in a public key will have the public key (or a cryptographic hash of it) in its name and a program principal (a measured Cloudproxy Hosted System) will have the measurement in the principal name (i.e.-a cryptographic hash).

The root name for a hosted program, in the development case, might look something like

```
key([0801100118012245080112410 46cdc82f70552eb...]).Program([25fac
93bd4cc868352c78f4d34df6d2747a17f85...])
```

Here, `key([0801100118012245080112…])` represents the signing key of the host and `Program([25fac93bd4cc868352c78f4d34df6d2747a17f85...])` extends the host name with the hash of the Hosted System[3]. If the host were a Linux host rooted in a TPM boot, its name would include the AIK and the PCRs of the booted Linux systems, the hash of the Authenticated Code Module ("ACM") that initiated the authenticated boot and the hash of the Linux image and it's initramfs[4]. In the section on SimpleExample execution output, there are many more examples of Tao Principal names.

The Cloudproxy API

The Cloudproxy programming model is simple and requires only a few API calls. Cloudproxy provides a programming interface in Go or C++ and we refer to the collection of Cloudproxy API calls as the "Tao Library."

There are two Cloudproxy principal API sets of interest for programmers. The first is the Tao API to access functions in a Hosted System. The interface definition is in go/tao/tao.go. The interface functions are:

```
GetTaoName()returns the Tao principal name of the Hosted System.

ExtendTaoName(subprin auth.SubPrin irreversibly extends the Tao
Principal Name by adding an additional node to the the
hierarchical Tao Principal Name.

GetRandomBytes(n int) returns n cryptographically secure random
bytes.

Rand() returns and io.Reader for random bytes from this Tao.

Attest(issuer *auth.Prin, time, expiration *int64, message
auth.Form) returns an Attestation from the Host System.  The
(optional) issuer, time and expiration will be given default
```

[3] As indicated by the ellipsis ("…") principal names are often longer and may even contain, say the full modulus of an RSA public key.

[4] Initramfs will have security critical code like the service that implements the Tao so it must be measured along with the kernel image to provide an accurate identity for the "running Linux OS."

```
values if nil; the message attested to is a form in the Tao
authorization language.

Seal(data []byte, policy string) returns a blob encrypted and
integrity protected by the Host System containing the data, policy
and measurement of the Hosted System in a form suitable for
Unseal.

Unseal(sealed []byte) decrypts and returns the data and policy
string, if access complies with the policy which usually includes
having the embedded Hosted System measurement match the embedded
measurement.
```

like names, path to key blobs, path to policy key, and the guard employed for authorization decisions. There are a few additional Tao functions, commonly used by Hosted Systems related to domain management and communications:

```
CreateDomain(cfg DomainConfig, configPath string, password
[]byte) (*Domain, error)

DomainLoad is used to store and retrieve Program Certificates and
sealed policy data.

Parent()which gets the parent interface to the Tao.  If t :=
tao.Parent(), for example, we'd call Attest as t. Attest(issuer,
time, expiration, message.

DialWithKeys(network, addr string, guard tao.Guard, v
*tao.Verifier, keys *tao.Keys).

Listen(network, laddr string, config *tls.Config, g tao.Guard, v
*tao.Verifier, del *tao.Attestation).
```

The Tao Library also contains helper functions to build and verify Program Certificates, perform common crypto tasks like key generation and establish the Tao Channel. Again, these are most easily understood by looking at the code example below.

Finally, the Tao Library has rather extensive and flexible authorization support. Authorization decision are performed by *guards* make.

Current guards include:

- The liberal guard: this guard returns true for every authorization query
- The conservative guard: this guard returns false for every authorization query
- The ACL guard: this guard provides a list of statements that must return true when the guard is queried for these statements.

- The datalog guard (used in the example below): this guard translates statements in the CloudProxy auth language (see tao/auth/doc.go for details) to datalog statements and uses the Go datalog engine from github.com/kevinawalsh/datalog to answer authorization queries. See install.sh for an example policy.

A brief description of the guards and authorization language appears in appendix 2 but you don't need to understand the authorization language to understand simpleexample.

Examples of all these calls (and their arguments), except for Rand, appear in the simpleexample code.

The second API is the Host API used by Host Systems and defined in go/toa/host.go. It consists of the following calls:

```
GetRandomBytes(childSubprin auth.SubPrin, n int) which returns
random bytes

Attest(childSubprin auth.SubPrin, issuer *auth.Prin, time,
expiration *int64, message auth.Form) which requests the Host
System's Host sign a statement on behalf of

Encrypt(data []byte) returns and encrypted, integrity protected
blob only the Host can decrypt.

Decrypt(encrypted []byte) returns the data protected by Encrypt if
this is the right Host.

AddedHostedProgram(childSubprin auth.SubPrin) notifies host that a
new Hosted System has been created.

RemovedHostedProgram(childSubprin auth.SubPrin) notifies the Host
that a Hosted System has been terminated.

HostName() returns the Principal Name of this Host System.
```

The best way to learn it is by looking at the annotated code in go/apps/simpleexample using the commentary below.

The Tao Paradigm

The Tao is almost always used in a stereotypical way which we refer to as the Tao Paradigm. Cloudproxy programs always have policy public keys embedded (PK_{policy}) in their image either explicitly or implicitly[5]. Statements signed by the corresponding private key (pK_{policy}), and only

[5] Explicitly embedding the key just means that it appears in initialized data measured as part of the program. An example of implicit embedding, which is described in more detail below, is reading in, say, the policy key and extending the identity of the program with that key.

those statements, are accepted as authoritative and acted on by these programs. The policy key(s) plus the Hosted System code and configuration, reflected in its measurement, fully describe how the Hosted System should behave and, hence, an authenticated measurement is a reliable description of expected behavior.

In the Tao Paradigm, when a program first starts on a Hosted System, it makes up a public/private key-pair ($PK_{program}$/ $pK_{program}$) and several symmetric keys that it uses to "seal" information for itself. A Hosted System then "seals," using the Host System interface, all this private (key) information[6]. After that, the Hosted System requests an Attestation from its Host System, naming the newly generated $PK_{program}$ and sends the resulting Attestation to a security domain service which confirms the security properties in the Attestation and Host Certificate[7]. If the Attestation and Host Certificate meet security domain requirements, the security domain service signs (with pK_{policy}) an x509 certificate specifying $PK_{program}$ and the Tao Principal Name of the Hosted System. The resulting certificate, called the *Program Certificate*, can be used by any Hosted System to prove its identity to another Hosted System in the same security domain. Program Certificates are used to negotiate encrypted, integrity protected TLS-like channels between Hosted Systems (the "Tao Channel"); Hosted System can share information over these channels with full assurance of the code identity and security properties of its channel peer. Once established, each endpoint of the Tao Channel "speaks for" its respective Hosted System.

Hosted Systems in the same security domain can fully trust other authenticated Hosted Systems in that security domain with data or processing. Typically, a Hosted System uses the symmetric keys it generates and seals at initialization to encrypt and integrity protect information it stores on disks or remotely.

Employing a centralized security domain service eliminates the need for each and every Cloudproxy Hosted System in a security domain to maintain lists of trusted hardware or trusted programs and simplifies distribution, maintenance and upgrade.

Often, Hosted Systems in the same security domain will share intermediate keys used to protect data that may be used on many Host System environments. As discussed below, when software is upgraded or a new Hosted System in a security domain is added, these keys can be shared based on policy-key signed directives as Host or Hosted Systems are upgraded or new systems are introduced in a controlled but flexible way eliminating the danger that data might become inaccessible if a particular Cloudproxy system is replaced or becomes damaged or unavailable.

Hardware roots of Trust

[6] The sample code in go/apps/fileproxy demonstrates rollback protection for this sealed data and we plan to improve local rollback support in the near future.

[7] The actual attestation being signed by the Host System expressed in a formalized language is $PK_{program}$ speaksfor the Hosted System Principal name.

Cloudproxy requires that the lowest level system software (the “base system”) be measured by a hardware component which also provides attest services and seal/unseal services and some hardware assist to isolate Hosted Systems. Absent hardware protection, remote users have no principled way to trust the security promises (isolation, confidentiality, integrity, verified code identity) since “insiders” might silently change security critical software or steal keys.

Cloudproxy supports TPM 1.2 and TPM 2.0 as hardware roots of Trust for Host Systems booted on raw hardware. We have implemented support for other mechanisms and believe adding a new hardware mechanism is relatively easy. In addition, Cloudproxy also can initialize and run on a “soft Tao” which simulates secure base system protection (but is not secure). This allows for easy debugging.

Once the base Host System is safely measured and booted on a supported hardware, Cloudproxy implements support for recursive Host Systems at almost every layer of software including:

1. A Host System consisting of hardware (e.g. - TPM, SMX) that hosts a VMM which isolates Hosted Systems consisting of Virtual Machines.
2. A Host System running in an operating system which isolates Hosted Systems consisting of processes (or applications).
3. A Host System running in an operating system which isolates Hosted systems hosted consisting of subordinate Operating Systems or Containers.
4. A Host System running in an application (like a browser) which isolates Hosted Systems consisting of sub-applications, like plug-ins.

In all cases, Hosted Systems have the same Tao interface to the Host System and can use any non-Cloudproxy host service (for example, any system call on Linux) so the programming model at each Hosted System layer is essentially unchanged from the non-Cloudproxy case.

Sample Applications

This paper is intended to allow you to use Cloudproxy immediately on a Linux based Cloudproxy Host System. To this end we include installation instructions for running under a “soft Tao” and TPM 1.2 based Tao protected hardware with SMX extensions. We also include the annotated code for a simple application called, cleverly, SimpleExample.

There are more complex examples in go/apps. Mention demo.

Installing Cloudproxy

First, you should download the Cloudproxy repository from [2]. To do this, assuming you have git repository support, type

git clone https://github.com/jlmucb/cloudproxy,

or,

go get https://github.com/jlmucb/cloudproxy.

This latter command will also install the needed go libraries. You can also download a zipped repository from github. You should probably install this in ~/src/github.com/jlmucb (which we refer to as $CLOUDPROXYDIR) to save go compilation problems later. It's a good idea to put go binaries in ~/bin as is common. Follow the installation instructions in $CLOUDPROXYDIR/Doc. That directory also contains [1] and an up to date version of this document as well as installation instructions for TPM 2.0 capable machines and installation for a Cloudproxy enabled KVM hypervisor and Docker containers.

You must also install the Go development tools (and C++ development tools if you use the C++ version) as well as protobuf, gtest and gflags as described in the Go documentation.

Next, compile, and initialize the SimpleExample application in $CLOUDPROXYDIR/go/apps/SimpleExample and run it as described in the next section.

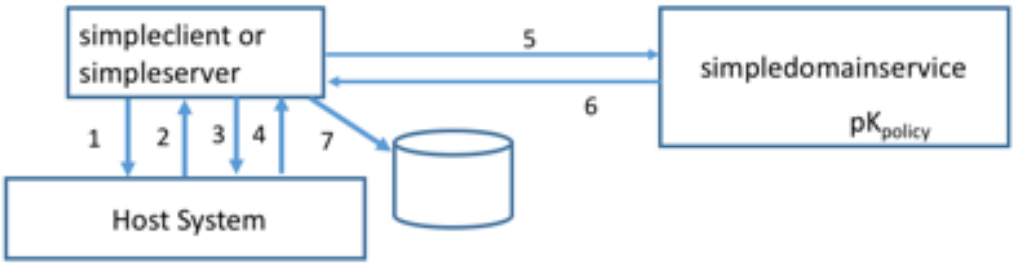
Understanding Simple Example

There are three application components in SimpleExample, each producing a separate executable:

1. Simple Client (in $CLOUDPROXYDIR/go/apps/SimpleExample/SimpleClient simpleclient.go)
2. Simple Server (in $CLOUDPROXYDIR/go/apps/SimpleExample/SimpleServer/simpleserver.go)
3. Simple Security Domain Signing Service (in $CLOUDPROXYDIR/go/apps/SimpleExample/SimpleDomainService/simpledomainservice.go)

Common code used by the client and server is in $CLOUDPROXYDIR/go/apps/SimpleExample/taosupport.

When *simpleclient* and *simpleserver* start for the first time on the Host System, they provide Attestations to *simpledomainservice* and, if the measurements are correct, *simpledomainservice* signs their respective Program Certificates with pK_{policy}). This interaction is depicted below.

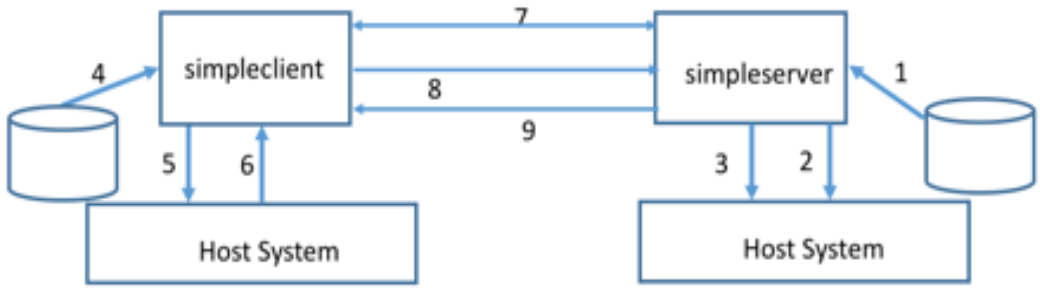


Initialization

1. simpleclient (or simpleserver) generates public/private key pair $PK_{simpleclient}$, $pK_{simpleclient}$. simpleclient requests Host System attest $PK_{simpleclient}$.
2. Host System retuns attestation
3. simpleclient generates additional symmetric keys and request Host System seal symmetric keys and $pK_{simpleclient}$.
4. Host System returns sealed blobs.
5. simpleclient connects to simpledomainservice and transmits attestation.
6. simpledomainservice returns signed Program Certificate.
7. simpleclient stores sealed blobs and Program Certificate for later activations.

Initialization

After initialization of keys and secrets, *simpleserver* makes up a secret waits for *simpleclient* to request their secret. Each *simpleclient* instance uses a Tao Channel to contact the *simpleserver* to learn the secret. We don't implement rollback protection or distributed key management for intermediate secrets in SimpleExample just to keep the example as simple as possible. *SimpleDomainService* is the domain service for SimpleExample.



Operation

1. Simpleserver reads previous sealed blobs and Program Certificate.
2. Simpleserver requests Host System unseal blobs yielding symmetric keys and private program key.
3. Host system returns unsealed blobs.

4. Simpleclient reads previous sealed blobs and Program Certificate.
5. Simpleclient requests Host System unseal blobs yielding symmetric keys and private program key.
6. Host system returns unsealed blobs.

7. Simpleclient and simpleserver open encrypted, integrity protected channel using their program keys and certificates.
8. Simpleclient transmits a request to retrieve secret.
9. Simpleserver retruns secret.

Operation

We describe the Tao API, compilation and installation, execution and output of the Go version of SimpleExample in the sections below. We also provide annotation for all the SimpleExample code containing all the critical Cloudproxy elements to help you get used to the programming model. Since the domain service does not use Tao primitives directly, we don't annotate that code here although `$CLOUDPROXYDIR/go/apps/SimpleExample` contains a full working version. A corresponding version of the annotated C++ version appears in Appendix III.

Although SimpleExample is very simple, the Tao relevant code in SimpleExample can be used with little change even in complex Cloudproxy applications.

Simple Client in Go

simpleclient is implemented in a single go file in go/apps/simpleexample/simpleclient/simpleclient.go that uses some common Tao based code in go/apps/simpleexample/taolibrary.

Parse flags, read configuration information, TaoParadigm, OpenTaoChannel, send a simple request, get response, print secret.

Simple Server in Go

Main: Parse flags, clear TaoParadigmData, TaoParadigm call server.

Server: Set up cert chain approval (NewCertPool, AddCert),
EncodeTLSCert(&serverProgramData.ProgramKey), Listen,
Loop: Accept, conn.(*tls.Conn).Handshake(),peerCerts :=
conn.(*tls.Conn).ConnectionState().PeerCertificates, peerCert :=
conn.(*tls.Conn).ConnectionState().PeerCertificates[0].Raw,
clientName = peerCert.Subject.OrganizationalUnit[0],
ms := util.NewMessageStream(conn),
go serviceThead(ms, clientName, serverProgramData),
serviceThread: Loop, taosupport.GetRequest(ms),
HandleServiceRequest(ms, serverProgramData, clientProgramName, req)

```
HandleServiceRequest: secret := clientProgramName + "43"
if *req.RequestType == "SecretRequest"
req.Data = append(req.Data, []byte(secret))
taosupport.SendResponse(ms, req)
```

Some Common code in Go

TaoParadigm: simpleDomain, err := tao.LoadDomain(*cfg, nil),
derPolicyCert := simpleDomain.Keys.Cert.Raw,
policyKeyName := sha256.Sum256(derPolicyCert)
hexPolicyCert := hex.EncodeToString(policyKeyName[0:32])
tao.Parent().ExtendTaoName(auth.SubPrin{e})
LoadProgramKeys(*filePath)
tao.Parent().Unseal(sealedSymmetricKey) or
InitializeSealedSymmetricKeys(*filePath, tao.Parent(),
SizeofSymmetricKeys)
SigningKeyFromBlob(tao.Parent(),sealedProgramKey, programCert,
delegation)or InitializeSealedProgramKey(
*filePath, tao.Parent(),
*simpleDomain)
FillTaoProgramData(derPolicyCert, taoName.String(),
*programKey, symKeys, programKey.Cert.Raw, filePath)

```
InitializeSealedProgramKey(filePath string, t tao.Tao, domain
tao.Domain) (
                *tao.Keys, error): CreateSigningKey(t)
```

```
RequestDomainServiceCert("tcp", *caAddr, k, domain.Keys.VerifyingKey)
should be replaced by RequestTruncatedAttestation

k.Delegation = na
        pa, _ := auth.UnmarshalForm(na.SerializedStatement)
        var saysStatement *auth.Says
        if ptr, ok := pa.(*auth.Says); ok {
                saysStatement = ptr
        } else if val, ok := pa.(auth.Says); ok {
                saysStatement = &val
        }
sf, ok := saysStatement.Message.(auth.Speaksfor)
kprin, ok := sf.Delegate.(auth.Term)
auth.Bytes(kprin.(auth.Bytes))
k.Cert, err = x509.ParseCertificate(newCert)
programKeyBlob, err := tao.MarshalSignerDER(k.SigningKey)
sealedProgramKey, err := t.Seal(programKeyBlob, tao.SealPolicyDefault)
ioutil.WriteFile
delegateBlob, err := proto.Marshal(k.Delegation)



tao.RequestTruncatedAttestation(network, *ca, keys,
domain.Keys.VerifyingKey)

OpenTaoChannel: x509.ParseCertificate(programObject.PolicyCert),
tao.EncodeTLSCert(&programObject.ProgramKey),
tao.DialWithKeys
tao.Listen(network, serverAddr, conf, g, domain.Keys.VerifyingKey,
keys.Delegation)
peerName := policyCert.Subject.OrganizationalUnit[0],
ms := util.NewMessageStream(conf)
        return ms, &peerName, nil

GetRequest, SendRequest, GetResponse, SendResponse
Protect, Unprotect
```

Configuring, compiling and running SimpleExample

When the Tao Host System starts, it requires several kinds of information:

- A public key that roots the *Tao* on the hardware,
- Host data, including the mechanism used to communicate between the Hosted System and the Host System, rules affecting which Hosted Systems the Host System should

run, and, in the case of a hardware rooted Host System, the hardware mechanism that is employed (e.g., TPM 1.2 or TPM 2.0).
- Domain data (in our case for the simpleexample domain) including the policy key and corresponding private key, and the self signed policy cert.

In addition, we need an implementation for the "Host System" which incudes support for the host provided isolation mechanism and communications channels used to communicate with Hosted Systems. In our case, the Host System is Linux and the implementation (whether using a soft tao, TPM 1.2 or TPM 2.0) is linux_host[8].

The public key rooting the hardware tao is usually produced by a tpm utility; in the TPM 1.2 nomenclature, this is called the AIK. The public key rooting the TPM 2.0 is the endorsement key. In our demo, we use a "soft tao" which is rooted in a key.

The Host Data consisting of keys and Host Certificate (used to validate nested Host System Attestation), and are in `$CLOUDPROXYDIR/go/apps/SimpleExample/SimpleDomain/domain.simpleexample/linux_tao_host`.

The Domain data including the policy key and corresponding private key, hostname, host type and communications channel, information related to the guards used[9] as well as signatures over the binaries that are part of the domain (if the Host System limits what Hosted Systems it will rum, in our case, these are the *simpleclient* and *simpleserver* binaries).

In simpleexample, all these information files in `$CLOUDPROXYDIR/go/apps/SimpleExample/SimpleDomain/domain.simpleexample`. Other sub-directories of `$CLOUDPROXYDIR/go/apps/SimpleExample/SimpleDomain/domain.simpleexample`, namely, `SimpleClient`, `SimpleServer` and `SimpleDomainService` contains data files stored and retrieved by these programs (like sealed keys and Program Certificates).

There is a single utility, called *tao* which initializes this domain data, activates the tao host and runs the applications. We provide shell scripts to call *tao* with the right arguments, these scripts are in SimpleDomain.

The scripts use several path variables, namely:

```
TAO_HOST_DOMAIN_DIR=~/src/github.com/jlmucb/cloudproxy/go/apps/simpleexam
ple/SimpleDomain
OLD_TEMPLATE=$TAO_HOST_DOMAIN_DIR/domain_template.simpleexample
DOMAIN=/Domains/domain.simpleexample
TEMPLATE=/Domains/domain_template.simpleexample
BINPATH=~/bin
```

[8] linux_host is also the implementation used by a KVM Host System and Docker containers and in fact, all Host Systems running on Linux..
[9] Look at domain.go for further details.

In addition, we need a generic domain template. We have provided a sample template in *SimpleDomain/domain_template.simpleexample*. However, you can generate such a template by running *gentemplate*, which consists of:

```
$BINPATH/tao domain init -tao_domain $DOMAIN -config_template $TEMPLATE -
pass "xxx"
/home/jlm/src/github.com/jlmucb/cloudproxy/go/run/scripts/domain_template
.pb > $TEMPLATE
sed "s/REPLACE_WITH_DOMAIN_GUARD_TYPE/Datalog/g"
```

This template contains information included in the policy cert, the basic datalog rules used by the domain when authenticating images and the location of the images which must me measured and included in the policy database in SimpleDomainService.

First, we must initialize the directory that will hold domain information. We do this by first

```
mkdir /Domains
```

and then calling initidomainstorage which consists of:

```
#
source ./defines
if [ -e $DOMAIN ]
then
  ls -l $DOMAIN
else
  mkdir $DOMAIN
fi
cp $OLD_TEMPLATE $TEMPLATE
source ./defines
if [[ -e $DOMAIN/SimpleClient]]
then
  echo "$DOMAIN/SimpleClient exists"
else
  mkdir $DOMAIN/SimpleClient
  echo "$DOMAIN/SimpleClient created"
fi
if [[ -e $DOMAIN/SimpleServer]]
then
  echo "$DOMAIN/SimpleServer exists"
else
  mkdir $DOMAIN/SimpleServer
  echo "$DOMAIN/SimpleServer created"
fi
if [[ -e $DOMAIN/SimpleDomainService]]
then
  echo "$DOMAIN/SimpleDomainService exists"
else
  mkdir $DOMAIN/SimpleDomainService
  echo "$DOMAIN/SimpleDomainService created"
fi
```

To initialize the (soft) key, call initkey which does the following:

```
#
source ./defines
if [[ -e $DOMAIN/linux_tao_host ]]
then
```

```
  echo "$DOMAIN/linux_tao_host exists"
else
  mkdir $DOMAIN/linux_tao_host
  echo "$DOMAIN/linux_tao_host created"
fi
KEY_NAME="$($BINPATH/tao domain newsoft -soft_pass xxx -config_template
$TEMPLATE $DOMAIN/linux_tao_host)"
    echo "host_name: \"$KEY_NAME\"" >> $TEMPLATE
```

"`newsoft`" means generate a new soft key. The arguments following the flags "`-config_template -tao_ -pass`" specify respectively the location of the template, the location where the domain information is stored and the password protecting the private policy key. This produces the xxx file containing root Tao key. If using the tpm, you'd call a corresponding program to put the AIK in template.

To initialize the domain, call initdomain which does the following:

```
#
source ./defines
$BINPATH/tao domain init -tao_domain $DOMAIN -config_template $TEMPLATE -
pub_domain_address "127.0.0.1" -pass xxx
$BINPATH/tao domain policy -add_host -add_programs -add_linux_host -
add_guard -tao_domain \
      $DOMAIN -pass xxx -config_template $TEMPLATE
```

The first call produces the files in $DOMAIN/linux_tao_host/{cert,keys,host.config}. The second measures the applications in the domain.

To initialize the (Linux) host, call inithost which does the following:

```
$BINPATH/tao host init -tao_domain $DOMAIN -hosting process -root -pass
xxx
```

This generates linux host configuration information which is in SimpleDomain/domain.simpleexample/linux_tao_host. The argument to the "`-hosting`" flag is the kind of child hosts, namely, Linux processes. The "`-root`" flag means this is a "root" host (i.e. – the lowest level tao). For hosts stacked on other hosts, we would use the "`-stacked`" flag. For example,

```
$BINPATH/tao host init -tao_domain $DOMAIN -hosting process -stacked -
parent_type tpm
```

To run the host, call runhost, which consists of:

```
$BINPATH/tao host start -tao_domain $DOMAIN -host linux_tao_host/ -pass
xxx &
```

The argument to the "`-host`" flag is the subdirectory of SimpleDomain/domain.simpleexample that contains the host information.

Finally, to run a Hosted System, like simpleclient, we would say:

```
$BINPATH/tao run $BINPATH/simpleclient -tao_domain $DOMAIN &
```

We have provided an additional script, "*runall*" which starts all the Hosted Systems and SimpleDomainService.

To summarize, to run simpleexample the very first time, call *initkey*, *initdomain* and *inithost*. If no host is running, call *runhost*. Each time you run tests call *runall* but remember to kill these services afterwards.

Some further observations: The password supplied in the calls to `tao domain init` and `tao domain policy` protect access to the policy private key. The password supplied to the `tao newsoft`, `tao host init` and `tao host run` protect the soft host private key; this password is usually not the same as the key protecting the policy private key. `linux_host` which implements the Linux Host System and as the shell scripts are configured, uses the rules generated by `tao domain policy` to decide whether to run an application. Normally, the Host System does not use application security domain rules to determine what to run and, in fact, usually will run any application. This can be accomplished with the inclusion of the rule:

```
LinuxHost(x)
(forall P: forall T: forall H: TrustedHost(T) and LinuxHost(H) and
Subprin(P, T, H) implies TrustedLinuxHost(P))
(forall P: Program(P))
(forall P: forall Q: forall H: Program(Q) and TrustedLinuxHost(H) and
Subprin(P, H, Q) implies MemberProgram(P))
```

where x is the measurement of the linux_host or an “AllowAll” policy. As you become familiar with the Datalog rules, you can apply them flexibly but that is a distraction in SimpleExample.

What the output from SimpleExample teaches us about the Tao

The most concrete way to understand Cloudproxy is to follow the code example and the output. Here is a brief description of the output of the Go version of SimpleExample using a “soft” tao. In the execution setup, the domain information is in `/Domains/domain.simpleexample`; this includes the template, tao prepared configuration files and three directories: `SimpleClient`, `SimpleServer` and `SimpleDomainService` which are directories in which application information (mostly sealed keys) are stored for, respectively, `SimpleClient`, `SimpleServer` and `SimpleDomainService`. Binaries are stored in the directory `~/bin` as is customary in go.

In the repository, there are also three shell scripts to facilitate running the examples. The script compile compiles the applications and puts them into bin. After making the directory, `/Domains`, use *initdomainstorage* to initialize the storage areas. Copy the script clean into `/Domains/domain.simpleexample/SimpleDomain` and make it executable.

Thereafter, modify any code you wish to and run *compile* in `$CLOUDPROXYDIR/go/apps/SimpleExample/SimpleDomain` to compile the programs. ***Then*** *as root*, run *runall* to run SimpleExample. After it runs, you can run *clean*, in `/Domains/domain.simpleexample/SimpleDomain`, to erase the output files. *clean* runs a ps aux | fgrep simple at the end to tell you what lingering processes to kill (kill -9) so you can run subsequent tests.

Our example uses the Datalog authorization subsystem so system rules are expressed in the Datalog policy language. Example statements in Datalog can be seen in the template file.

When you look at the output, you'll notice, at the beginning:

```
Warning: Passwords on the command line are not secure. Use -pass option
only for testing.
Warning: Passwords on the command line are not secure. Use -pass option
only for testing.
Warning: Passwords on the command line are not secure. Use -pass option
only for testing.
Warning: Passwords on the command line are not secure. Use -pass option
only for testing.

Linux Tao Service (key([0801100118012245080112410431 0f3c0d7c5ff1...]))
started and waiting for requests

2016/02/20 11:27:13 simpledomainservice: Loaded domain
2016/02/20 11:27:13 simpledomainservice: accepting connections
```

This indicates that the linux_host, simpledomainservice, simpleclient and simpleserver have been initialized. The second section shows that the linux_host for the soft tao (with the indicated key) has started. The final section indicates that the domain service is started and waiting for request.

Next, you'll notice,

```
TaoParadigm: my name is
key([08011001180...]).Program([94d80d932fbc...]).key(f3169de17b1032dde2
30423f7d11dde89c143de147188fa67acf613d63da0420)
```

This is from *simpleserver*, and it is the Tao Principal Name of your *simpleserver* program, running on your Host System, after it has been extended with the hash of the loaded policy certificate. If you look at the source code for *simpleserver*, you'll notice that the policy key is not embedded in the code; if it had been, the policy key would be reflected in the program measurement. Instead, we read in the policy key cert and extend the *simpleserver* Tao Principal Name with the hash of the self-signed policy cert. The Tao Principal Name is hierarchical. The first segment, "`key([08011001180...])`", describes the host root[10]. The second segment, "`Program([94d80d932fbc...])`", describes the *simpleserver* program reflecting its measurement. The third segment, "`key(f3169de17b1032dde230423f7d11dde89c143de147188fa67acf613d63da0420)`", describes the policy key as noted above. Observe that the Tao Principal Name fully reflects all the program code as well as the policy it will execute (as represented by the policy key).

For the rest of this description, we will simplify terms like "`Program([94d80d932fbc...])`" as "`Program(`*`program-measurement`*`)`".

[10] If the root host had been a TPM, the name would include the TPM's AIK, and the contents of PCR 17 and 18 which contain the measurement of the booted Linux, extended with the initramfs which contains all the security critical files used by the Linux instance

Next, notice the statement:

```
simpledomainservice, speaksfor: key(simpleserver_program_key) speaksfor
key(host-key).Program(simpleserver-measurement).key(policy-key)
```

This is the statement that TaoParadigm will use to request an attestation from the Linux Host System. The resulting Host System supplied attestation is

```
key(host-system-key) from notBefore until notAfter says [simpleserver-
program-certificate] speaksfor key(linux-host).Program(simpleserver-
measurement).key(policy-key)
```

This statement is sent to the domain service which, after checking the measurements and domain policy signs a certificate (with pK_{policy}) that includes the statement

```
key(policy-key) from notBefore until notAfter says [simpleserver-
program-certificate] speaksfor key(linux-host).Program(simpleserver-
measurement).key(policy-key)
```

This is the *simpleserver* Program certificate. *Simpleserver*, as we described in the code annotations, stores this certificate, and sealed versions of the corresponding private *simpleserver* ProgramKey and SymmetricKeys. Decrypted and useable versions of these keys are populated in *serverProgramData* by *TaoParadigm*.

After initialization, *simpleserver* waits for client connections.

simpleclient meanwhile, goes through the same *TaoParadigm* initialization (which is not duplicated here) obtaining its Program certificate. *ServerClient* calls *OpenTaoChannel* with it's Program Cerificate and corresponding key. You'll notice, later in the output, that *simpleserver* opens a secure channel with a peer

```
key(linux-host).Program(simple-client-measurement).key(policy-key)
```

That peer is just your *simpleclient*. Normally, the Host System on which *simpleclient* runs will be different from the one *simpleserver* runs although in our case, they run on the same host.

Finally, you'll notice that *simpleserver* receives a request

```
2016/02/20 11:27:16     message type: 1
2016/02/20 11:27:16     request_type: SecretRequest
```

and returns the secret which is received by simpleclient as

```
simpleclient: secret iskey(linux-host).Program(simpleclient-
measurement).key(policy-key)43
```

simpleclient encrypts and integrity protects the secret with its symmetric keys and the process concludes.

We have only discussed the major output elements here. Your output will contain much more including log messages from *simpledomainserver*.

The certificate for the simpleclient (which is in `/Domains/domain.simpleexample/SimpleClient/signerCert)` is:

```
Certificate:
    Data:
```

```
        Version: 3 (0x2)
        Serial Number: 1455996433984 (0x15300267640)
    Signature Algorithm: ecdsa-with-SHA256
        Issuer: C=US, O=CloudProxy, OU=, ST=WA, CN=SimpleExampleTest
        Validity
            Not Before: Feb 20 19:27:16 2016 GMT
            Not After : Feb 20 19:27:16 2017 GMT
        Subject: C=US, O=Google,
OU=key([0801100118012245080112410431 0f3c0d7c5ff1490ace20f167e7de1d5c6847c
84d498c3b4a8087031b49d9a38e7e59f4c5e4f23adc6ce2e394c7ac48923bcfcd7446bba0
f86ef8bbdf89b6d5]).Program([d38d94100ae2bb57cccb97cb347ab060fb28c382bafe1
38f253fd19b491b1a15]).key(f3169de17b1032dde230423f7d11dde89c143de147188fa
67acf613d63da0420), ST=, CN=localhost
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:7a:d5:40:f6:80:fd:73:a5:80:b8:88:57:7c:60:
                    6d:87:b6:78:4a:3f:fc:1c:cc:40:af:34:2d:98:31:
                    02:21:02:71:65:66:7f:90:49:91:88:91:21:43:c7:
                    f5:50:de:0a:7c:58:c8:6c:10:06:46:fc:3c:1a:a1:
                    bb:c7:20:c6:83
                ASN1 OID: prime256v1
        X509v3 extensions:
            X509v3 Key Usage: critical
                Key Agreement, Certificate Sign
            X509v3 Extended Key Usage:
                TLS Web Server Authentication, TLS Web Client
Authentication
    Signature Algorithm: ecdsa-with-SHA256
         30:44:02:20:0c:a0:99:55:79:0d:b7:26:20:07:38:03:da:ba:
         ff:28:0c:fd:94:f6:4e:5f:b1:ad:41:11:89:42:61:fd:5b:e7:
         02:20:7f:26:62:ee:2a:4c:90:e4:f4:7c:d6:c6:2c:b6:1d:db:
         d8:4a:bc:b9:60:26:aa:80:e8:bf:74:bd:ee:34:cb:fe
```

You'll notice that `/Domains/domain.simpleexample/SimpleClient/` also contains the files sealedsigningKey (*simpleclient's* sealed program private key), retrieved_secret (the "secret" encrypted with *simpleclient's* symmetric keys) and sealedsymmetricKey (*simpleclient's* sealed symmetric keys).

Running SimpleExample on a TPM1.2 machine

TODO

Upgrade and key management scenarios

Since sealed material is only provided to a Hosted System with exactly the same code identity that sealed the material running on the exact same Host System, while isolated by that Host System, you may be worried about lost data when a Hosted System breaks or becomes unavailable or limitations that may affect key management, software upgrade or distribution when the Hosted System runs on other Host Systems. In fact, it is rather easy to accommodate

all these circumstances, and many others, efficiently, securely and in most cases automatically using Cloudproxy, although Cloudproxy applications must make provisions for this during development.

Below are a few sever example key management techniques that can be used when a Cloudproxy application is upgraded, a new Cloudproxy application (in the same security domain) is launched, or as applications migrate to other Host Systems. All these mechanisms preserve the confidentiality and integrity of all Cloudproxy applications and their data.

There is a discussion of many of the mechanisms, as they might affect client software used across different security domains, by users with no control over the application code while supporting consumer transparency (the most challenging case) in [4]. Here we restrict ourselves to cooperating server applications for simplicity.

To ease description, imagine all application data is stored locally or remotely and probably redundantly in encrypted, integrity protected files. Each file is encrypted and integrity protected with individual file keys and each file key is itself encrypted and integrity protected with a group sealing keys. Different groups of file keys are protected by different sealing keys to reduce the risk of universal compromise. Every key has exposed meta data consisting of a globally unique name for the entity it protects, the key type and an "epoch." Epochs increases monotonically as the keys are rotated[11]. As keys for a new epoch become available, the objects they protect are re-encrypted, over a reasonable period of time (the Rotation Period). During this time, keys for the prior epoch are available and can be used to decrypt objects; however, as soon as new epoch keys are available, all new data is encrypted with the new epoch keys. At the end of the Rotation Period, once applications have confirmed that all data is protected with the keys from the most recent epoch, old epoch keys are deprecated.

The first option to deal with "brittle keys" protecting application data is standard: use a distributed key server like Keyczar (or many others). In this case, Cloudproxy applications do not locally store data protection keys but contact a key server (over a Tao Channel). The key server (which does key rotation, etc., as many do) authenticates the Hosted System that needs keys and verifies that it is authorized to receive those keys; if so they are transmitted over the Tao Channel. Hosted Systems can be upgraded and all authorization policy can be maintained by the key service. Hosted Systems will need to respond to "reinitialize" requests periodically as keys rotate.

An alternative, less centralized, key rotation mechanism allows individual Hosted Systems maintaining their own keys to protect files as well as perform key rotation themselves. When software is upgraded or new programs are introduced, the new programs or upgraded programs come with a certificate signed by the policy key that instruct one Hosted System to disclose these keys to the new version (or new) Hosted System. Since this can result in lost data if a Host System becomes unavailable, Hosted Systems would likely distribute these keys to different instances on different machines to ensure continuity.

[11] And you certainly should rotate keys as part of effective cryptographic hygiene!

Finally, when new data protection keys are established for an application task, Hosted Systems can contact a domain service to receive intermediate keys for registered files or file classes. These keys can be sealed using the Host System provided Seal and used without contacting the service each time the Hosted System starts. This mechanism places additional administrative burden on each Hosted System to contact the "key sharing service" as intermediate keys rotate but this is not uncommon.

It is important to note that while the foregoing descriptions treat keys as "all or nothing" entities, all these scheme have corresponding "split key" implementations to achieve higher security. In addition, any security domain may elect to have an authorized Cloudproxy Hosted System archive data. Such an archive application, upon which security domain policy confers access to data, can, in the background, archive data to (centralized or distributed) repositories.

As a reminder: there are other possible mechanisms to do key management.

Finally, note that application upgrade (given a data key management solution) is automatic even when the policy keys change: New versions of Hosted Systems simply re-initialize (get new program keys and certificates) using the (centralized or distributed) security domain service and no special provision, aside from current policy at the security domain service, need be provided[12].

Suggested Exercises

That's all there is to using Cloudproxy. Here are some suggested exercises to complete the training:

1. Write a more complicated set of domain applications; for example, see "`$CLOUDPROXYDIR/go/apps/fileproxy`."
2. Boot a Linux Host System on tpm supported hardware using the TPM to root the Linux Tao (see … for instructions).
3. Boot a KVM Host System on tpm supported hardware and then run a stacked VM host in a Linux partition (see … for instructions). SimpleExample should run fine in the VM(s) with slight changes to the initialization scripts.
4. Explore the Data log engine (examples?)
5. What happens if you make a modification to simpleclient.go and immediately run it on `linux_host` without reinitializing the domain?

References

[12] Many events may cause such a policy change including a determination that previously trusted hardware elements have been compromised.

[1] Manferdelli, Roeder, Schneider, The CloudProxy Tao for Trusted Computing, http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-135.pdf.
[2] CloudProxy Source code, http://github.com/jlmucb/cloudproxy.
[3] TCG, TPM specs,
http://www.trustedcomputinggroup.org/resources/tpm_library_specification
[4] **Beekman, Manferdelli, Wagner,** Attestation Transparency: Building secure Internet services for legacy clients**.** AsiaCCS, 2016.

Cloudproxy Guards

The Guard interface:

- `Subprincipal() auth.SubPrin`: returns a unique subprincipal for this policy.
- `Save(key *Signer) error`: writes all persistent policy data to disk, signed by key
- `Authorize(name auth.Prin, op string, args []string) error`
- `Retract(name auth.Prin, op string, args []string) error`
- `IsAuthorized(name auth.Prin, op string, args []string) bool`
- `AddRule(rule string) error`
- `RetractRule(rule string) error`
- `Clear() error`: removes all rules.
- `Query(query string) (bool, error)`
- `RuleCount() int`
- `GetRule(i int) string`.
- `String() string`: returns a string suitable for showing auth info.

CloudProxy's Authorization Language

SimpleExample in C++

Simple Client in C++

Simple Server in C++